FuzzSlice: Testing ability to prune false positives of differing error classes

Lucas Fox

CS5130

Introduction

Static analysis programs produce many false positives when analyzing a codebase. This leads to developers having to spend more time diagnosing which reported bugs are real issues and which ones are not. To make matters worse, many developers admit they often ignore reports generated by static analysis tools because of the decently high false positive rate.

To help fight this and help developers better prioritize resources, University of Waterloo researchers created FuzzSlice to help alleviate this pain, a tool designed to prune false positives efficiently. While many modern tools aim to confirm true positives, FuzzSlice aims to take the smallest slice compilable that is encapsulating the bugs and fuzz it to determine if it is a true bug or something else.

Fuzzing is a common tool used in this situation. By throwing random input at a program or function, one can expect bugs to present themselves through crashes. However, thorough fuzzing can be a computationally expensive and time-consuming process, especially when fuzzing an entire program. FuzzSlice tries to address this limitation by fuzzing the smallest "slice" of code possible, thus limiting the total lines of code, dependencies, and execution time.

FuzzSlice utilizes Address Sanitizer, referred to as ASAN in this report, to detect buffer overflow bugs while fuzzing. While this is obviously effective for buffer overflow bugs, it means ASAN may be missing many the ability to detect many bugs of various types, including uninitialized memory bugs and bugs with undefined behavior. To combat this, we use a combination of ASAN, Undefined Behavior Sanitizer (UBSAN), and Memory Sanitizer (MSan), later explained in methodology, to detect these different bugs and increase coverage. While not a goal of this project, an altered fuzzing time is also tested and can be reasonably used to compare efficiency of different fuzzing execution times.

Methodology

In order to improve FuzzSlice's detection ability, we attempt to use a multi-layer strategy for testing rather than using a single sanitizer/oracle. A technical challenge in this is that some sanitizers cannot be used at the same time—Msan and ASAN in this case. This is because they both require the same exclusive access to mock memory that tracks application state and runtime. Because of this, these sanitizers cannot be combined and ran simultaneously, but rather back-to-back or on separate machines.

We created the following sanitizer testing strategy:

1. **ASAN + UBSAN:** This was the configuration that was expected to have the highest coverage and best results. When combined, these sanitizers can detect both memory issues and undefined behavior (often logic errors) at the same time. Whereas an undefined error would be never be detected in the original FuzzSlice testing, the fuzzer will now raise flags for those crashes.
2. **MSan Only:** Once again, memory sanitizer cannot be run at the same time as address sanitizer. This was tested to see if MSan would detect bugs that were not compilable or not reachable by ASAN. Usage and interpretation of this sanitizer will be discussed in results.
3. **UBSAN Only:** UBSAN only was tested to get a baseline understanding of logic errors and differences between UBSAN and ASAN in detection.
4. **ASAN Only:** ASAN only was also tested so we could compare it to other combinations and sanitizers. The FuzzSlice technical documentation identifies erroneous and erratic bugs in the testing repositories, so we decided to exclude those. Because of this, we needed to run ASAN only on the bugs we cared about so we could properly compare.

ASAN+UBSAN	*"-fsanitize=fuzzer,undefined,address -fno-sanitize-recover=undefined,address"*
MSan Only	"-fsanitize=fuzzer,memory -fno-sanitize-recover=memory"
UBSAN Only	"-fsanitize=fuzzer,undefined -fno-sanitize-recover=undefined"
ASAN Only	"-fsanitize=fuzzer,address -fno-sanitize-recover=address"

Table 1: A table displaying combination sanitizers and their flags

As mentioned above, an altered testing time was also used. The original FuzzSlice technical documentation notes that full runs of testing on all datasets with 5 minutes of fuzzing on premium hardware (64 Core processor, 128 GB DRAM) took ~3-4 days to run. Due to this, we elected to test on a much lower time: 5 seconds of fuzzing. We are confident that we will get similar efficiency and results to the original paper as the authors of FuzzSlice were very confident that FuzzSlice could detect bugs from crashes very swiftly, and that 5 minutes was meant to be exhaustive. 5 seconds of fuzzing in FuzzSlice should be highly efficient due to the minimal slices of code. We're also confident in saying that if a crash is not detected in 5 seconds of fuzzing, it probably won't be detected in 5 minutes of fuzzing. Instead, that computation time and resources are better spent testing the next bug.

Implementation

This project was implemented on the supplied Docker image supplied by the author of FuzzSlice. Using the *noblematthews/fuzzslice* docker image, we needed to modify the target bugs for each repository. This is so we could test on all relevant bugs and exclude bugs with identified issues.

We also modified *config.yaml* and *main.py. config.yaml* is where we can modify the fuzzing time and library/repo we're testing, among other things. In *main.py,* on line 691, we can modify the compiler command (Table 1) to select our sanitizer combination.

```
test_library: tmux
fuzz_tool: 0
bug_timeline_targets_run: false
timeout: 5
hard_timeout: 30
max_length_fuzz_bytes: 15000
parallel_execution: false
crash_limit: 1000
log_report: true
```

Figure 1: config.yaml

For datasets, we are testing the datasets the original paper tested on. These are the opensource repositories *tmux, oppenssh-portable,* and *openssl*. While the Juliet dataset was tested in the original paper and included in the docker image, disjunction between the *info_lib* and *test_lib* directories seems to be an issue that we couldn't resolve in time.

Evaluation and Results

To compare these sanitizer combinations, we tested the relevant bugs in their respective repositories with the previously discussed setting changes.

As predicted, **UBSAN + ASAN** proved to flag more potential false positives than **ASAN Only.** **MSan Only** also flagged more bugs than **ASAN Only**, meaning there is value in running MSan separately despite the limitation in ASAN and MSan running simultaneously. Included are result logs of modified FuzzSlice being run on tmux:

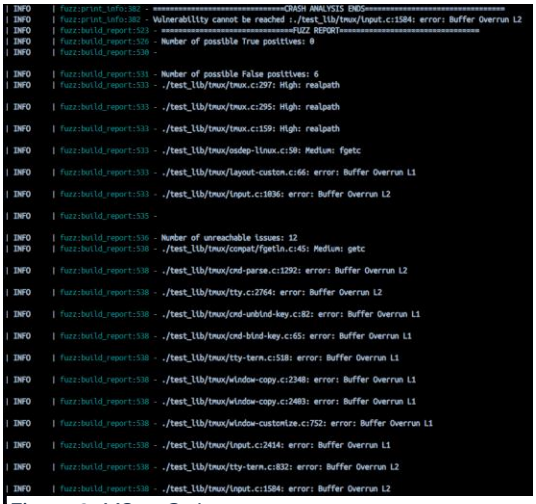

Figure 2: MSan Only

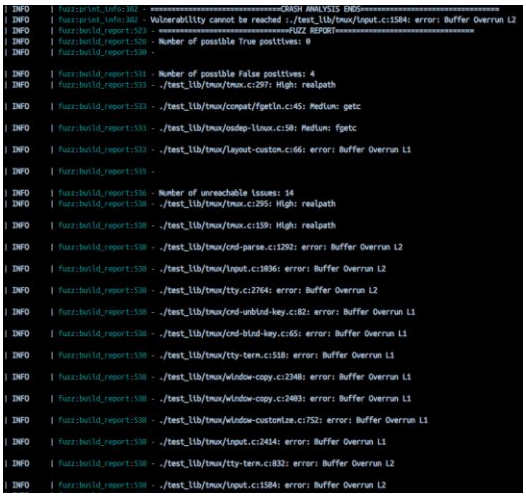

Figure 3: ASAN Only

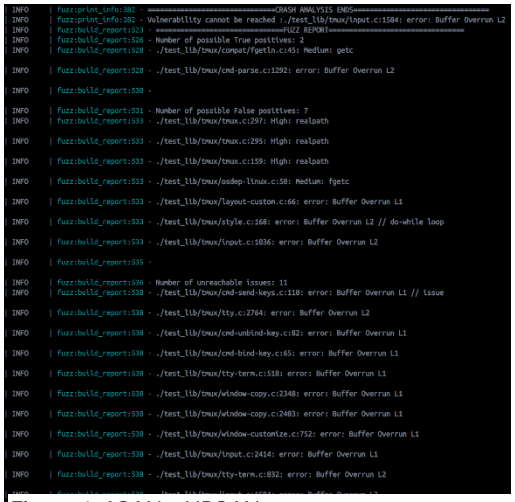

Figure 4: ASAN + UBSAN

Conclusion

To conclude, this project found that the direct fuzzing utilized in FuzzSlice can be further improved by using a combination of different sanitizers (multi-layered) rather than single sanitizers. We can also conclude that there is value in running both Msan and UBSAN + ASAN, as MSan was able to detect uninitialized memory issues that were silent in other tests. FuzzSlice was able to detect a wider range of possible false positives, better allowing developers to prioritize which bugs flagged by static analysis programs they should focus on and triage.

While the runs still did take a decently long time (minutes to hours), we also indirectly found that we can aggressively decrease the fuzzing time and still get similar results to running longer fuzzing times. This backs up our idea that bugs will fail fast, and that the resources are better spent attempting to get other bugs to fail fast as well.